Scalable Composition, Evolution and Verification Through Feature-Oriented Programming

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu
(contact author)

Kathi Fisler
Worcester Polytechnic Institute
kfisler@cs.wpi.edu

Don Batory
University of Texas at Austin
batory@cs.utexas.edu

October 30, 2001

Abstract

A growing trend in software construction advocates a change in system modularity. While traditional modules permit easy re-configuration of a system to support different *actors*, these new modules encapsulate *features*. These modules better match the language of requirements. As a result, programmers find it easier to design, compose and evolve systems. We have demonstrated, through theory and experiment, that these benefits extend to software verification also. This program now requires work on programming language and environment design and implementation, type systems, interface languages, and more refined verification techniques. Our long-term goal is to build on these successes to carry feature-oriented system construction to maturity, through collaboration and cross-pollination between software engineering, programming languages and formal verification.

1 Introduction

Effective methodologies for constructing and evolving large-scale software systems remain the holy grail of software engineering. Ideally, an effective construction methodology should support reuse, configurability, evolution, and validation techniques for checking requirements. Attractive ideas along these lines include components (for encapsulating reusable code fragments), product-line architectures (for building families of systems from components), formal analyses (for determining properties of components), and predictable assembly (for deriving properties of a system from properties of its components). While these techniques have identified key pieces of the software construction puzzle, none of these is inherently sufficient to address the numerous software construction problems that beg resolution. Furthermore, integrating these pieces is often difficult because they don't quite align in their assumptions, style and approaches.

A fundamental problem with many current approaches is that they view systems from the perspective of *producers*, rather than *consumers*. Software consumers tend to specify requirements primarily in terms of features; the actors in the code, in contrast, often reflect internal implementation details. (For example, consider a GUI toolkit: users care about editing, searchability and copying features, but toolkits export classes pertaining to windows, panels, canvases and so forth.) Recent research in software construction thus increasingly reflects a common theme: *we need to realign our modules around features rather than actors*. This shift in perspective must be informed by input from at least three communities: software engineering, which tackles mapping user needs to concrete requirements; programming languages, which addresses the mapping of requirements to code; and formal verification, which provides validation support at each stage.

Feature-oriented models suggest a world in which we select modules that implement the features that a system needs, then compose those modules to build the system. Research into feature-oriented modules demonstrates that they enable reuse, evolution, maintenance, product line construction, configuration, and validation better than conventional modules [2, 10, 16]. This evidence strongly suggests that feature-oriented modules will form a cornerstore of future software development technologies. A shift to feature-oriented modules, however, has far-reaching effects in how we design, view, and use modules. These differences form the focus of our research agenda.

This paper argues for feature-oriented modules as a foundation for scalable, robust, software construction. We present an overview of feature-oriented modules, survey their technical characteristics and established benefits, and outline the open problems that we must solve in order to build a software construction methodology around them.

2 Feature-Oriented Modules: An Overview

Traditional modules encapsulate participants (or *actors*) and contain the code that the actor needs to implement services (or *features*) of the system. Feature-oriented modules, in contrast, encompass all of the code needed to implement a single feature (or closely related set of features) across the actors in a design. In other words, feature-oriented modules *cross-cut* actors. Figure 1 depicts the difference between actor-oriented and feature-oriented modules.

By effecting a change in focus from producers to consumers, feature-oriented modules offer several advantages over actor-orientation:

- They make it easier for programmers to satisfy requirements, because requirements primarily describe features. Actor-oriented modular decompositions are difficult to adapt to changing requirements, and don't easily yield product lines. As a result, they are often hard for end-users to understand and use.

- They simplify configuration, evolution, and maintenance because these activities are often driven by user demands. Consider a simulator for military missions. The simulator could contain personnel and weapons as actors, and missions for firing on targets as features. In order to be useful for training military units for a range of missions, the simulator would need to be easily configurable along many axes. For example, consider a case where a division needs to train to use a particular set of weapons in a particular terrain. A simulator exists for the right terrain, but it includes some inapplicable weapons. How hard would be it be to remove the missions (communication protocol and firing decisions) for those weapons?

  With feature-oriented modules, each module encapsulates code for a mission centered around a particular weapon under a certain set of conditions. To remove the weapons, the programmer removes the modules for those weapons from the design and composes the simulator afresh. With an actor-oriented modularization, in contrast, a programmer would need to edit each actor involved in negotiating for or using those weapons. As

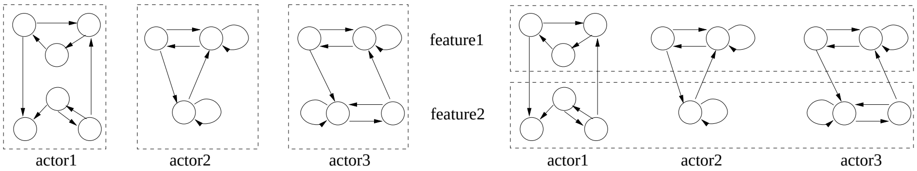

Figure 1: Two modularizations of the same code: actor-oriented (left) and feature-oriented (right). This example depicts code as state machines to simplify the illustration.

actor-oriented code usually co-mingles code for different features, this task would be time-intensive and error-prone. In many cases, code modification is not even possible; for example, a programmer may be trying to extend proprietary code that is only available as object code.

- They simplify formal validation. Ideally, we should be able to exploit the modular decomposition to verify properties of programs. Reasoning about properties modularly is extremely attractive because some forms of formal verification suffer from a problem known as *state-explosion*: the models generated during verification are often too large to be verified. In this case, verification engineers must decompose the verification problem into smaller (hopefully tractable) problems. In the process, engineers attempt to isolate the portion of the model that affects the property.

  Since properties arise from requirements, which in turn correspond closely to features, a feature-oriented decomposition makes it easy to isolate the relevant modules. With actor-oriented modules, in contrast, the feature-oriented property cross-cuts the actors. The verification engineer often must decompose the property statement itself into statements about the individual actors. Experience shows this step to be difficult, if not impossible. Feature-oriented modules therefore support a more natural modular verification methodology.

Many researchers have built systems using a feature-oriented decomposition for diverse domains; a brief sample includes protocol stacks [6, 21], testbench generators [14] and verification tools [20]. Some researchers have abstracted these implementations into feature-oriented architectures, including *refinements* [4], *collaboration filters* [5], *units* [10], *aspects* [15], *collaborations* [17], *hyper-slices* [18] and others. Not all of these approaches, however, view their cross-cutting fragments as modules in the theoretical sense (with interfaces and composition semantics).

We view the much-heralded notion of aspect-oriented programming [15], for instance, as raising an important issue of system decomposition. The state of current aspect technology, however, falls short in various respects. Aspects suffer from several problems that inhibit practical deployment: they lack modular compilation, modular verification and composition validation techniques. Nevertheless, many "aspect-oriented" problems fit extremely well in a feature-oriented framework. We therefore view feature-oriented modules as providing much of the expressive power of aspects, with just enough restriction to enable the other activities of the software engineering enterprise.

The Structure of Feature-Oriented Modules

Typically, several actors collaborate to implement a feature. If a module encapsulates all of the code pertaining to a single feature, a feature-oriented module must contain code fragments for several actors (as shown in Figure 1). This affects both the content and the interfaces of modules.

In Java, for instance, a package encapsulates fragments of classes: a class extension within a package may extend a class outside the package. The package, however, explicitly names the other packages whose contents it imports. This results in a completely determined class hierarchy. Packages, therefore, primarily organize code, and offer no flexibility of reconfiguration.

Feature-oriented modules also encapsulate class extensions. The modules differ, however, in stating only the interfaces of their imports. A (potential) third-party is responsible for composing the modules by satisfying module imports with other modules that implement the desired interfaces. This *external linkage* property forces programmers to make meaningful modules that a third-party can sensibly combine.

The external linkage of modules trickles down to their contents. Whereas in Java, the superclass of a class extension is easy to determine (by following the module's import), in a feature-oriented decomposition, the superclass won't be known until linkage. Classes with parameterized super-classes are called *mixins*;[1] their compilation techniques, type theory and so forth are open topics of research [7, 13, 19, 22].

What about constraints between the modules? Features in a system are rarely orthogonal. In a telephony system, for example, one cannot have conference calls without the ability for multi-party connections. Some systems require the opposite situation, where one feature disallows another; for example, we cannot easily add a feature for computing undirected spanning trees to a graph with a directed edges feature. This enriches the interfaces of feature-oriented modules: they must specify *design rules* that govern when a feature can be integrated into a system.

Richer modular analyses demand even more interface information. In order to reason about a feature in isolation from the rest of the system, techniques such as model checking [8] must determine exactly how the actors start executing a feature. For example, do the actors begin to execute the feature simultaneously, or does one actor start the feature and then convince the others to follow? Module interfaces must provide this form of *feature coordination* information.

3 Prior Results

Our belief in the effectiveness of feature-oriented modules stems from extensive prior experience. We have applied them to constructing monolithic software and software product lines for military command-and-control simulators [3], databases [4], programming environments [9], verification tools [12], and other systems. Our models of feature-oriented design have emerged from these experiences. Batory and his colleagues have implemented support for expressing feature-oriented designs directly in both Java [1] and C++ [19].

We have also implemented lightweight and heavyweight formal methods techniques for these modules. Batory and Geraci introduced and provided tool support for design rule checking [1]. Flatt, Krishnamurthi and Felleisen have developed a type system and compilation methodology for a mixin-centric adaptation of Java [13]. Fisler and Krishnamurthi present a technique for model checking feature-oriented modules [11]. They show that feature-orientation promises to eliminate some of the property decomposition and circularity problems that dog existing modular verification techniques. We have since implemented a model checker for these modules, and applied it to a portion of the military command-and-control simulator to validate our claims; we have even found feature-oriented modules naturally control state space growth in other ways as well [16]. Our prior papers provide extensive overviews of results by other researchers in this area.

4 Objective and Vision

Our objective is to carry a feature-oriented design methodology to maturity. We believe this will make it easier for programmers to satisfy requirements, help them compose software more effectively, provide better guarantees of module and system behavior, and adapt better to requirement and code evolution.

To realize this vision, we anticipate solving several technical problems:

- We need to develop programming language support for the types, interfaces and linking mechanisms that feature-orientation requires.
- We must design type systems that capture design-rule checking; this raises interesting problems in representing negative information.
- We must refine and expand the content of interfaces to support more validation techniques.
- We need to expand our stable of verification techniques; model checking is good for verifying control properties, but less suitable for data-intensive properties.
- We also need to consider how to manage requirement evolution.
- Finally, we need to build integrated programming environments that support all of these techniques.

We believe, however, that these problems are tractable, and that solving them will enable us to build on the promising foundation that feature-oriented modules provide for large-scale software construction.

[1] Note that these are completely unrelated to the Common Lisp notion of *mixins*, which are a pattern of programming with multiple-inheritance.

References

[1] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, Feb. 1997.

[2] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *International Conference on Software Reuse*, June 2000.

[3] D. Batory, C. Johnson, B. MacDonald, and D. von Heeder. FSATS: An extensible C4I simulator for army fire support. In *Workshop on Product Lines for Command-and-Control Ground Systems at the First International Software Product Line Conference (SPLC1)*, August 2000.

[4] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, Oct. 1992.

[5] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, Oct. 2001.

[6] E. Biagioni, R. Harper, P. Lee, and B. G. Milnes. Signatures for a network protocol stack: A systems application of Standard ML. In *ACM Symposium on Lisp and Functional Programming*, 1994.

[7] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, Mar. 1992.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001. To appear.

[10] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *ACM SIGPLAN International Conference on Functional Programming*, pages 94–104, 1998.

[11] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Symposium on the Foundations of Software Engineering*, Sept. 2001.

[12] K. Fisler, S. Krishnamurthi, and K. E. Gray. Implementing extensible theorem provers. In *International Conference on Theorem Proving in Higher-Order Logic: Emerging Trends*, Research Report, INRIA Sophia Antipolis, September 1999.

[13] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[14] Y. Hollander, M. Morley, and A. Noy. The *e* language: A fresh separation of concerns. In *Proceedings of TOOLS Europe*, Mar. 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

[16] H. Li, K. Fisler, and S. Krishnamurthi. The influence of software module systems on modular verification. In review, Oct. 2002.

[17] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Mar. 1999.

[18] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717), IBM, Apr. 1999.

[19] Y. Smaragdakis and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.

[20] K. Stirewalt and L. Dillon. A component-based approach to building formal-analysis tools. In *International Conference on Software Engineering*, 2001.

[21] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. Technical Report 97-1638, Department of Computer Science, Cornell University, July 1997.

[22] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 1996.